# Argonauts: A Working System for Motivated Cooperative Agents

Daniel Hölzgen, Thomas Vengels, Patrick Krümpelmann,
Matthias Thimm, Gabriele Kern-Isberner

Information Engineering Group,
Technische Universität Dortmund,
Germany

**Abstract.** This paper presents the ARGONAUTS multi-agent framework which was developed as part of a one year student project at Technische Universität Dortmund. The ARGONAUTS framework builds on a BDI approach to model rational agents that act cooperatively in a dynamic and indeterministically changing environment. However, our agent model extends the traditional BDI approach in several aspects, most notably by incorporating motivation into the agent's goal selection mechanism. The framework has been applied by the ARGONAUTS team in the 2010 version of the annual multi-agent programming contest organized by Technische Universität Clausthal. In this paper, we present a high-level specification and analysis of the actual system used for solving the given scenario. We do this by applying the GAIA methodology, a high-level and iterative approach to model communication and roles in multi-agent scenarios. We further describe the technical details and insights gained during our participation in the multi-agent programming contest.

## 1 Introduction

Formal approaches for representing the mental state of an intelligent agent mostly employ the BDI model [3], a framework that originated in psychology and describes intelligent behavior such as decision making, deliberation, and means-end reasoning in rational beings. This model divides a mental state into *beliefs*, *desires*, and *intentions* and gives a formal account for their interactions. Beginning with the work of Rao and Georgeff [13] many researchers in the field of artificial intelligence and intelligent agents applied this (informal) framework to formalize autonomous intelligent behavior [16, 15]. Most modern computational frameworks implementing the BDI approach such as JADEX [12] or JASON [2] allow for a declarative specification of an agent and leave the actual execution of the agent to some BDI interpreter. This approach allows for a flexible representation that is easy to comprehend.

This paper deals with a working system that implements the BDI approach and makes use of many ideas and technologies developed for rational agents and multi-agent systems in the past years. This system has been realized during a one year lasting student project at the Technische Universität Dortmund [1].

The system is based on AgentSpeak [14] and builds on top of Jason [2], an interpreter for an extended version of AgentSpeak. In order to provide for a more sophisticated knowledge representation and reasoning facility the beliefs of the agents are represented as extended logic programs and inference is performed using the answer set semantics [8, 9]. We further benefit from this kind of representation by using the planning language $\mathcal{K}$ [5] which bases on logic programming as well and allows a tight integration of belief representation, reasoning and planning components. The system also incorporates motivations [11] into the agent's mental model, thus enhancing the agent's desire handling and goal selection mechanism.

The Argonauts framework has been applied in the annual multi-agent contest[1] which provided for a complex scenario and gave the opportunity to test, evaluate, and improve the developed framework. There, two opposing teams consisting of 20 agents each have to compete against each other in the task of cow herding in a discrete but incomplete and indeterministic grid environment. Due to the cows unpredictable behavior and with each agent having a limited field of vision it is necessary to perform this task in a cooperative manner. Besides obstacles the environment can contain fences which can be opened by an agent stepping on the corresponding switch field. While opening a fence for others an agent cannot walk through the fence by itself and more importantly such a switch field must not necessarily exist on both sides of a fence. Thus, complex plan construction is mandatory given the demanding maps of the contest. The team having the most cows in its own corral in average wins the match.

The rest of this paper is organized as follows. In Section 2 we give some necessary technical preliminaries. In Section 3 we give an overview on the abstract agent architecture and cooperation model employed in the Argonauts system. We continue in Section 4 with a brief description of the concrete system design and afterwards in Section 5 we present some highlights of the implemented system. We have a closer look at the scenario specific requirements for the multi-agent contest and the team strategy in Section 6. In Section 7 we discuss some technical details of the implemented system and we conclude in Section 8.

## 2 Preliminaries

In this section we give a brief overview on answer set programming [8, 9] which is used for knowledge representation and reasoning throughout the Argonauts system. Afterwards, we present the basics of both the AgentSpeak language [14] which is used as a foundation for our system and of the planning language $\mathcal{K}$ [5] which is based on logic programming. Finally, we give a brief overview on the GAIA methodology [17] for multi-agent system design.

[1] `http://www.multiagentcontest.org`

### 2.1 Answer Set Programming

We consider extended logic programs which distinguish between classical and default negation [8, 9] and use a first-order language without function symbols except constants, so let $\mathcal{L}$ be a set of literals, where a literal $h$ is an atom $A$ or a (classical) negated atom $\neg A$. The symbol $^-$ is used to denote the complement of a literal with respect to classical negation, i. e. it is $\overline{p} = \neg p$ and $\overline{\neg p} = p$ for a ground atom $p$.

**Definition 1 (Extended logic program).** *An* extended logic program $P$ *is a finite set of rules of the form* $r : h \leftarrow a_1, \ldots, a_n, \mathsf{not}\ b_1, \ldots, \mathsf{not}\ b_m$ *where* $h, a_1, \ldots, a_n, b_1, \ldots, b_m \in \mathcal{L}$. *We denote by* $head(r)$ *the head* $h$ *of the rule* $r$ *and by* $body(r)$ *the body* $\{a_1, \ldots, a_n, \mathsf{not}\ b_1, \ldots, \mathsf{not}\ b_m\}$ *of the rule* $r$. *The set of all extended logic programs is denoted by* $\mathcal{P}$.

If the body of a rule $r$ is empty ($body(r) = \emptyset$), then $r$ is called a *fact*, abbreviated $h$ instead of $h \leftarrow$.

Given a set $X \subseteq \mathcal{L}$ of literals, then $r$ is *applicable* in $X$, iff $a_1, \ldots, a_n \in X$ and $b_1, \ldots, b_m \notin X$. The rule $r$ is *satisfied* by $X$, if $h \in X$ or if $r$ is not applicable in $X$. $X$ is a model of an extended logic program $P$ iff all rules of $P$ are satisfied by $X$. The set $X \subseteq \mathcal{L}$ is *consistent*, iff for every $h \in X$ it is not the case that $\overline{h} \in X$. An answer set is a minimal consistent set of literals that satisfies all rules. This can be characterized as follows.

**Definition 2 (Reduct).** *Let* $P$ *be an extended logic program and* $X \subseteq \mathcal{L}$ *a set of literals. The* $X$-reduct *of* $P$, *denoted* $P^X$, *is the union of all rules* $h \leftarrow a_1, \ldots, a_n$ *such that* $h \leftarrow a_1, \ldots, a_n, \mathsf{not}\ b_1, \ldots, \mathsf{not}\ b_m \in P$ *and* $X \cap \{b_1, \ldots, b_m\} = \emptyset$.

For any extended logic program $P$ and a set $X$ of literals, the $X$-reduct of $P$ is a logic program $P'$ without default-negation and therefore has a minimal model. If $P'$ is inconsistent, then its unique model is defined to be $\mathcal{L}$.

**Definition 3 (Answer set).** *Let* $P$ *be an extended logic program. A consistent set of literals* $S \subseteq \mathcal{L}$ *is an* answer set *of* $P$, *iff* $S$ *is a minimal model of* $P^S$.

*Example 1.* Listing 1 shows an excerpt of an actual belief base of an agent in the cows and herders scenario. It models how an agent can decide wether to herd cows or scout to find new cows. Whenever the agent does not perceive any cow at a given point in time he knows he cannot herd and should scout.

### 2.2 AgentSpeak and Jason

AgentSpeak allows for the specification of agents by a set of beliefs and a set of plans. Both beliefs and plans are represented in a declarative language similar to logic programs. Compared to logic programs, beliefs are represented as facts, and plans look similar to rules, but with different syntax and semantics. Besides beliefs and plans another important concept is that of goals. In AgentSpeak, a goal is a state of the system an agent wants to become true. This can be thought

**Listing 1** A logic programming snippet for reasoning over actions

```
time(7).
percept(cow5,4).
percept(cow4,7).

see_cow :- percept(C,T), cow(C), time(T).

herd :- see_cow.
scout :- not see_cow.
```

of as a desire from classical BDI architecture an agent has chosen to become true. There are two types of goals, achievement goals and test goals. In the following we restrict our attention to achievement goals, see [2] for a full account in both achievement and test goals. In general, achievement goals $!g(t)$ describe states where a belief $g(t)$ becomes true.

In order to express how an agent can achieve a goal, plans are used. A plan consists of three elements: a triggering event, a context, and a body. A triggering event $e$ is an expression in the form $+b(t)$ or $-b(t)$, if $b(t)$ is a belief, or $+!g(t)$, $-!g(t)$ if $g(t)$ is a goal. The context is a formula $b_1, ...b_n$ over belief literals $b_i$. A plan body is a sequence $h_i$ of goals or actions, delimited by ";". So a plan is an expression of the following form:

$$e : b_1, ..., b_n \leftarrow h_1; ...; h_n.$$

Triggering events denoted by a "+" prefix refer to plans handling belief additions or achievement goals. A "−" prefix denotes plans handling belief loss or achievement failures.

Informally, the triggering event marks which plans an agent could adopt as an intention whenever it tries to react to the event. The context determines if a plan is consistent with an agent's beliefs, this allows an agent to apply different plans for handling the same event in different situations, based upon its beliefs. The plan chosen to handle an event is adopted as an intention. In Jason, each intention is a stack of statements from a plan body that are executed sequentially by the interpreter.

Another feature provided by Jason is the concept of *internal actions*. An internal action is a java class, and instances of those classes can be invoked directly from plans. The syntax resembles that of an atom, prefixed with a java package name, or a "." denoting the default library name space. Internal actions do not modify the environment, instead they can be used to alter an agent's internal state. Parameters are expressed as terms, and in case of variables, internal actions can be used as oracles to ground those variables beyond the possibilities provided by Jason's procedural reasoning engine.

*Example 2.* Listing 2 shows a simplified AgentSpeak plan which is used to declare the leader's behavior for driving cows. The triggering event $+!drivencow(C)$ marks that this plan is used to handle an achievement goal *drivencow*. As the

**Listing 2** An AgentSpeak plan for driving cows

```
+!drivencow(C) : true
 <-  ?ison(C,CX,CY,_)[source(percept)];
     ?corralcenter(OX,OY);
     argonauts.createDrivingPlan(CX,CY,OX,OY);
     !execDrivingPlan.
```

context of the plan is set to *true* there is no limitation for when the plan can be used. The agent will first determine the position of the cow $C$ and the position of its own corral. Afterwards an internal action is executed to generate a plan, which is added to the agent's plan library as a plan for the *execDrivingPlan* goal. Thus, the dynamically generated plan can be executed by adding another achievement goal in the last line of the plan.

### 2.3 Language $\mathcal{K}$

The language $\mathcal{K}$ [5] is a logic-based planning language used for describing transitions between states of knowledge. It is based on extended logic programs and makes use of default negation to deal with incomplete knowledge. In this section we give a brief overview on some key aspects of this language, see [5, 6] for a complete description and formal definition.

In $\mathcal{K}$, the *static background knowledge* of the planning problem is represented by a normal stratified logic program, defining the predicates that are not subject to change. Contrary to this, the *dynamic knowledge* consists of *fluents* and *actions* that are subject to change and are denoted via

$$p(X_1, ..., X_n) \texttt{ requires } t_1, ..., t_m$$

where $p$ is a positive action literal, $X_1, ..., X_n$ are variable symbols and $t_1, ..., t_m$ are type literals in form of positive literals of the static background knowledge. The main construct of $\mathcal{K}$ are *causation rules*, defining static and dynamic dependencies. This rules are of the form

$$\begin{aligned} &\texttt{caused } f \\ &\quad \texttt{if } b_1, ..., b_k, \texttt{not } b_{k+1}, ..., \texttt{not } b_l \\ &\quad \texttt{after } a_1, ..., a_m, \texttt{not } a_{m+1}, \ldots, \texttt{not } a_n \end{aligned}$$

where $f$ is a literal or $false$, $b_1, ..., b_l$ are fluents or types, and $a_1, ..., a_n$ are literals. These rules state that if the "`if`" part of the rule is known to be true in the current state, and the "`after`" part is known to be true in the previous state, then $f$ is known to be true in the current state. The default negation "`not`" can be used in both the "`if`" and the "`after`" part of the rule, to deal with incomplete knowledge similar to answer set semantics. To specify whether or not it is possible to execute an action, execution rules are specified as follows:

$$\texttt{executable } a \texttt{ if } b_1, ..., b_m, \texttt{not } b_{m+1}, ..., b_n$$

**Listing 3** A $\mathcal{K}$-plan for driving cows

```
fluents:   at(C,L) requires cow(C), location(L).
actions:   drive(C,L) requires cow(C), location(L).
always:    executable drive(C,N) if connected(N,L), at(C,L).
           caused at(C,L) after drive(C,L).
initially: at(cow1,location4).
           noConcurrency.
goal:      at(cow1,corral)? (4)
```

With $a$ being an action literal and $b_1, ..., b_n$ literals. This rule states that if the "`if`" part of the rule is known to be true in the current state, action $a$ can be executed. Besides these rules there are several other constructs such as *constraints* and *safety restrictions* which we omit here. After specifying the planning domain plans can be requested by querying goals of the form

$$g_1, ..., g_m \texttt{ not } g_{m+1}, ..., g_n \ ?\ (i)$$

where $g_1, ..., g_m$ are ground fluent literals forming the goal, and $i$ being the length of the requested plan. A plan is a sequence of action literals whose execution leads from an initial state to a desired goal state.

*Example 3.* A plan in $\mathcal{K}$ is shown in Listing 3. It demonstrates the basic idea for generating plans to drive cows into the own corral. Therefore, the cow's position is declared as a fluent and driving a cow from one location to another is declared as an action. The `executable` rule says that it is only possible to drive a cow if a connection exists between both locations, followed by a rule saying that after driving, the cow is located at the new location. The `goal` asks for a plan to locate the cow inside the own corral. The resulting plan must not contain concurrent actions, which is stated by `noConcurrency`.

### 2.4 The GAIA methodology

GAIA [17] is a high-level methodology for analysis and design of agent-based systems. It divides the development into an analysis- and a design phase with different levels of abstraction. During analysis, roles and their interactions are derived as abstract entities. These are used during design to develop concrete models of agents, the services they offer depending on the roles they implement, and the acquaintances among the different types of agents. We will now give a short overview of the different models shown in Figure 1.

First, the key-roles are identified and defined in a roles model, providing an abstract description of its function. A roles model defines the role's permissions to access all kinds of resources, its responsibilities and available activities, and interaction protocols. The interaction protocols are defined in an interactions model, describing the communication between roles. This includes which roles initiate and respond to the interaction as well as which information is used or supplied during the course of the interaction.

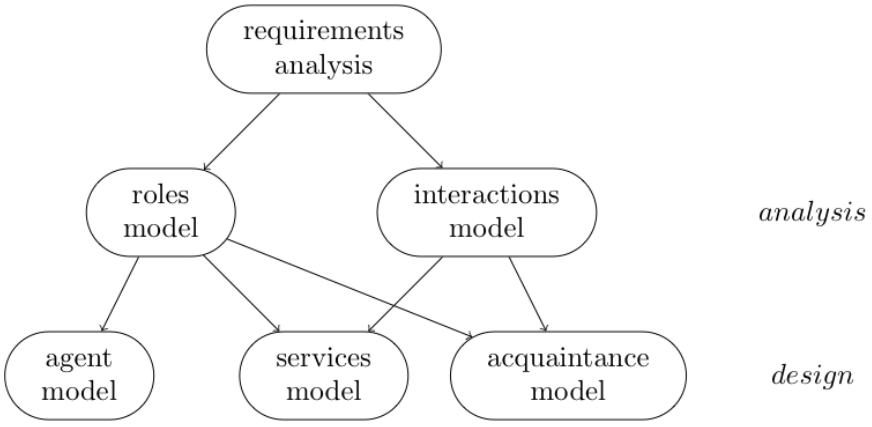


**Fig. 1.** Models in GAIA

Both the roles model and the interactions model are now used to derive the concrete models during design. The agent model describes the different agent types present in the system. It is defined which roles are implemented by an agent type and how many agents of this type exist. The services model identifies the services provided by each role and documents its properties in terms of inputs, outputs, pre-conditions and post-conditions. Finally, an acquaintance model defines the communication links between the agent types in form of a directed graph. It is primarily used to identify bottlenecks in the communication structure. Although the models developed during design-phase are concrete enough to have direct counterparts in the implemented system they are not meant to be directly implemented but rather act as a foundation so that traditional design methodologies can be applied.

## 3 System Analysis and Specification

In this section, we will first describe the cows and herders scenario provided by the multi-agent contest, followed by an analysis and specification of our multi-agent system. We did not follow a specific methodology for the analysis of requirements but observed requirements arising immediately from the given cows and herders scenario, and designed and implemented a cooperative system of rational agents accordingly. Therefore, our analysis consists of the following artifacts we investigate: the overall agent architecture, a cooperation model, and a scenario analysis. The agent architecture is presented by a formal model for describing the internal structure of individual agents. The cooperation model describes which steps an agent should undertake for solving problems in a group of agents. The scenario analysis results in necessary conventions like roles and

communication protocols each agent has to comply with. We start with a brief topical survey on the scenario and the task to be accomplished.

### 3.1 Cows and Herders

As shown in Figure 2, the environment used in the multi-agent programming contest is a grid-like world, containing cows and the agents of both teams. Beside empty cells and impassable obstacle cells, there are fences that can be opened by placing an agent on a cell adjacent to a switch cell which has to be part of every fence. Due to the possibility that these cells could be blocked by obstacles, it is possible that fences can be opened from only one side of the fence or not at all. The simulation is processed in discrete steps, giving every agent the opportunity to send its next action to the server, i. e. either to move towards one of the eight possible directions or stay at its current position.

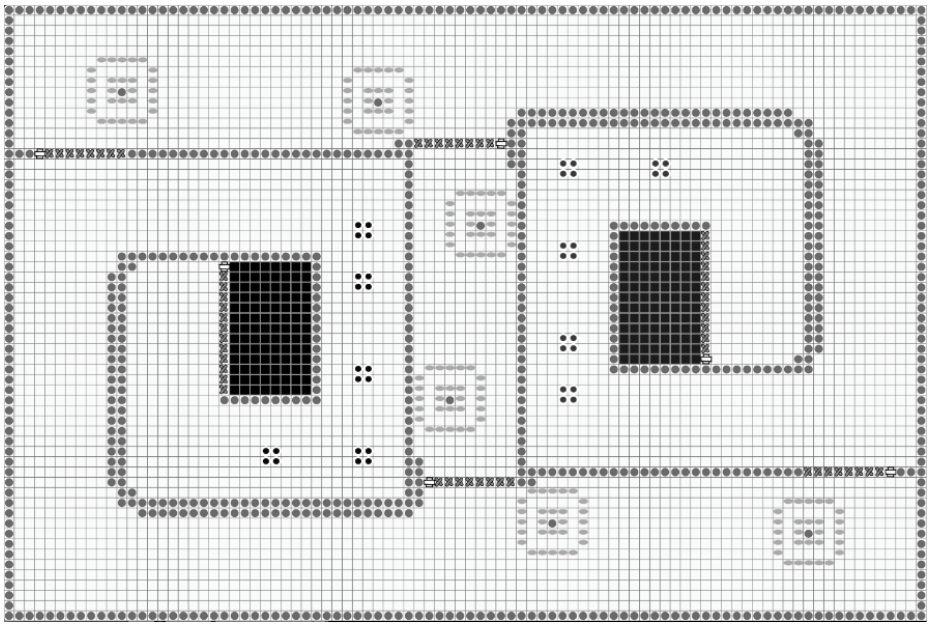

**Fig. 2.** A map used in the multi-agent programming contest

The behavior of the cows can only be controlled indirectly: They will move away from agents if these come close enough, and tend to prefer to move towards cells near other cows. Thus, by placing one or more agents in appropriate positions near the herd, the cows are likely to move in the desired direction. Since cows are quite fast, and due to their indeterministic behavior while moving, the task of driving cows should be done by a coordinated groups of agents.

The agents have a limited field of vision, in addition to this their perceptions might be false. In each step, every agent gets its perceptions in form of contents of the grid cells in its field of vision. However, communicating is not done through the environment, so every agent can communicate with any other agent of the team independent of the respective distance or simulation steps. At the beginning, all agents are provided with information about their own corral's position, which does not include any information about the fields surrounding it, so it is not necessarily clear in which way the corral can be accessed. At the end of the simulation, the team with the highest average amount of cows during the whole simulation wins the match.

### 3.2 Agent Architecture

We developed an agent architecture based on the BDI model [13]. In the BDI approach the mental model of an agent consists of the following three major components: *beliefs*, *desires* and *intentions*. Beliefs represent an agent's subjective beliefs about the world and itself. They should be consistent but not necessary objectively true. Desires describe world states the agent wants to become true and thereby reflect the wishes of the agent. Intentions comprise possible actions the agent is capable of and decided to commit. They can be viewed as sequences of actions, or plans, that are based on certain desires an agent decided to achieve as goals.

Since the desires have an important impact on the possible intentions that an agent will pursue, our approach focuses on refining the way desires are generated. Therefore, we extended our BDI model by a motivation component. Motives allow an agent to generate desires based upon its personality, and not only upon the current beliefs about the world. They are used to not only generate desires according to the current situation, but also to equip them by a degree of intensity that helps to select one as a goal. This degree of intensity can slowly be adjusted according to the agent's situation. This results in a less reactive and more autonomous way in which an agent acts.

Moreover, we also specified a planning component so that intentions are not only based upon static, predefined plans. Instead, a planner can generate a sequence of actions dynamically based upon the goal to be achieved and the environmental state an agent has in mind.

Figure 3 shows our extended BDI model with its components, and how an agent is connected to the environment. The given figure also shows process and information flow. Process flow is indicated by solid lines, i. e. solid lines indicate the order in which single components work with internal data and how single components depend on each other. Data flow is indicated by dashed lines, i. e. dashed lines indicate which information is relevant to and updated by single components. Below we give a brief description how the agent interacts with an environment, and how the internal structure works. The environment provides an agent with perceptions which in our case include both "visual" percepts and messages from other agents. From an external point of view the environment

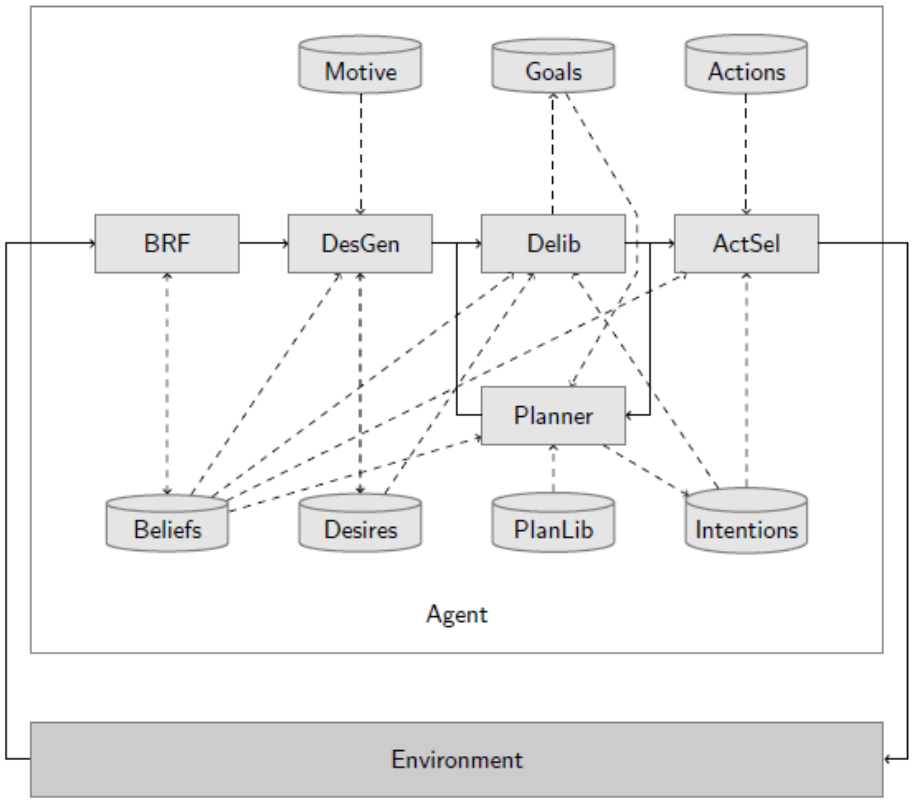


**Fig. 3.** The ARGONAUTS BDI model. Dashed lines indicate data flow, solid lines indicate process flow.

receives actions from an agent which can be both messages for communications or actions modifying the state of the environment the agent is situated in. The architecture of the agent itself is based on the BDI model, extended by a motivation-driven desire generation and a planning component. We describe the extended BDI cycle in more detail: First, the agent gets its perception from the environment, including information communicated by other agents. This information is used to revise the agent's beliefs. After this, desires are generated according to the agent's current situation and its motivation, representing objectives the agents would like to accomplish. In the next step, a deliberation component selects which desires should be selected as goals the agent will pursue in terms of concrete intentions. To generate the intentions, the deliberation component makes use of a plan library, or, if no existing plan is applicable, a plan generator to dynamically generate a plan in order to fulfill the selected goal. At the end of the cycle, an action selection component determines the action to

be executed by the agent, according to its current intention. In the following, we will give a simplified example on how the architecture works.

*Example 4.* Suppose that a single agent named `alice` is placed in a scenario similar to the cows and herders example. Its (relevant) current beliefs are as follows:

$$\texttt{cow(cow1)}.\ \texttt{iam(alice)}.$$
$$\texttt{ison(cow1,5,5,1)}.$$
$$\texttt{ison(alice,1,1,1)}.$$

Informally, the agent `alice` knows who it is, that there is a cow `cow1`, and knows where in the grid world it and the cow are located (on grid cells `(1,1)` and `(5,5)`, respectively). The last argument of the `ison` predicate represents the last time the position was updated and is used to support the revision of the beliefs. We additionally assume that `alice` currently has two desires, one to scout and one to drive `cow1` into the corral, with the scouting desire being currently annotated with a higher degree of intensity as the other desire. Now, the agent gets the percept `ison(cow1,4,4,2)`. Because the cow cannot be on both cells at the same time, the *belief revision function* will revise the old information about the cow with the new one.

After revising the agent's beliefs the agent's desires are generated. In order to win the scenario `alice` is equipped with a motive that aims at driving cows into the corral. Now that the cow has come closer the degree of intensity of this desire is raised. Thus, in the next step the deliberation component detects that the degree of intensity of the cow driving desire is now greater than the one to scout. A goal change is performed, selecting the cow driving desire as the new goal of the agent. The deliberation component now makes use of the planner which generates a plan for driving the cow. This plan is selected as the new intention of the agent and is refined until it consists of at least one atomic intention, in this example moving one step towards the cow. Those atomic intentions are precise enough to be executed by performing suitable actions which are selected by the action selection component. Here, the agent performs one step towards the cow.

### 3.3 Cooperation model

One important aspect of multi-agent systems is cooperation. In the cows and herders scenario, we are faced with a team of agents that has to compete against another team of agents. The reward of each game is given to a team, not to individual agents. Since each agent only plays for its team, agents should behave cooperative instead of self-interested, because they cannot gain individual rewards. Additionally, problems that agents encounter and have to solve are often too complex to be solved by a single agent. In this way, cooperation means that agents have to work together to solve a problem. Therefore, some formalism or guidelines are required how to specify and implement a cooperative agent.

In order to get a better understanding what cooperation means in the cows and herders scenario, consider the following example.

*Example 5.* Two agents, Alice and Bob, intend to cross a fence. Since fences are only crossable if an agent operates a switch associated with a fence, one agent has to stand next to the switch, for example Bob. This will cause the fence to open, but only Alice can now cross the fence because a switch operator, like Bob, cannot cross a fence while operating the switch to keep it open. After Alice passed the fence, the same procedure has to be repeated, but with Alice and Bob switching their roles.

The example above shows that certain tasks have to be evaluated and performed by more than one agent. When distributing a given problem solution among multiple agents, each agent can solve a particular part of the problem, which results in a team solution for problems completely unachievable by single agents.

We now describe how we specified cooperation among agents on an abstract level. To this end we adapted ideas from various planning paradigms presented in [16], and created a *phase model* that maps different stages of distributed planning to BDI cycles. Whenever an agent tries to achieve a goal that is only achievable by working together in a group, it performs the following steps:

1. **Grouping**: In a first step the agent sends a help request with a description of the goal that has to be achieved. Each agent of the team can now accept the request and become a helper, or reject the request. If too many agents accept the request, the initiator sends rejections to dispensable agents.
2. **Task Decomposition**: This phase consists of dividing a goal into subgoals. The division process continues until tasks are found that can be achieved by single agents. The subgoals found are assumed to be achievable in parallel. The division is done by the initiator of the previous grouping request.
3. **Task Allocation**: In this phase, the results of the task decomposition phase, i. e. the subgoals, are assigned to agents that agreed to cooperate in the grouping phase. Subgoals that arise directly from the same goal can either be independent, in which case the order of achieving a subgoal does not matter. Otherwise, subgoals have to be achieved in parallel by different agents, and therefore are assigned to different agents. This results in a distributed contribution of multiple agents for a solution, and therefore gives rise to a local cooperation of several agents of the team.
4. **Individual Planning**: In this phase, an agent creates a plan for a given goal from the previous task decomposition and allocation phases. This results in a sequence of actions the agent will commit to in order to achieve its goal.
5. **Conflict Resolution**: If conflicts arise due to the concurrent nature of working together and interdependencies between subgoals, the initiator, or leader, is responsible to provide a solution.
6. **Execution and Monitoring**: This phase refers to the final execution of single actions determined by a plan and the observance of effects of those actions. For conflicts that occur during plan execution, e. g. some position is blocked by a cow, an agent might choose to solve conflicts locally by choosing the next optimal cell to move to. Whenever a fatal error occurs an agent reports the error to the initiator. Then the initiator has to decide whether the task can still be achieved or if it is mandatory to re-plan.

Of course, these phases only describe planning and cooperation on a very abstract level. For tasks solvable by single agents on their own the same model is applied and both "grouping" and "parallel task allocation" phases are omitted.

### 3.4 Scenario Analysis

We analyzed the cows and herders scenario and used it for testing our agent system. Following the GAIA methodology mentioned in section 2.4, the analysis phase results in two major artifacts: a role model and an interactions model. The first step is to identify the key roles in the multi-agent system. This first role model is only a prototypical list of key roles with informal descriptions. We identified the following keys roles:

**Cowbot** The cowbot role defines all common interactions and attributes for all our agents. Every agent adopts this role which can be understood as a base role.

**Driver** The driver is responsible for driving cows towards a target location. A driver can drive a single cow or a herd of many cows. We assume that multiple drivers collaborate to drive cows.

**Scout** The scout role is adopted to explore the environment. Its main purpose is to find cows. Another important aspect implicitly addressed is the discovery of environmental details like walls or fences while searching for cows.

**Leader** The leader role is adopted by agents that are responsible for solving problems not already processed by agents, and allocating agents to tasks for solving a problem. This role involves the coordination of other roles.

**Door Opener** The door opener is responsible for opening fences. Therefore, the door opener has to operate the switch associated with a fence.

After identifying these key roles an interaction model is created. An interaction model contains all protocols relevant for the multi-agent system. Here, a protocol is a pattern of an interaction between various roles. While a coarse role model reflects the entities found in a system, an interaction model specifies which interactions occur between these roles. We identified the following protocols within our system:

**Broadcast Information** This protocol is used to relay relevant information that should become common knowledge, in particular perceptions and the positions of the agents are communicated via this protocol.

**Request Help** This protocol is used to ask other agents for help whenever a leader cannot accomplish a task on its own. The request is addressed to various roles and the respective agents.

**Task Monitoring** This protocol is used by agents executing a task cooperatively to inform their leader about the current progress. The protocol is not only used to allow a leader to control a plan progress, but also to assign new tasks to fulfill a goal if the situation changes.

After identifying the key roles and interactions within our multi-agent system, the role model can be finalized. This results in an elaborated role model in which key roles are described in detail with their attributes and protocols.

## 4 System Design and Architecture

In this section, we describe the design and architecture both of the framework for executing extended BDI agents and the MAS built on it, according to the insights gained during the analysis phase. We start with a more detailed description of the extended BDI architecture, followed by a closer look at the *belief revision* and *desire generation*. Afterwards, we will go into the design of the MAS, with a focus on interaction and communication.

### 4.1 The extended BDI architecture

As mentioned in Section 3, our agent is based on an extended BDI architecture. Its beliefs are represented as extended logic programs [9]. All percepts received by the environment or communications are transformed into facts that are used to revise the current beliefs about the world. Beside these, the agent is provided with base knowledge which can be both facts and rules represented in the form of extended logic programs. This can be useful to support adherence to conventions as well as to deduce additional knowledge, like the connection of switches to fences.

Based on the approach described by Meneguzzi and Luck [11], we decided to refine the way desires are generated by using motivation, to result in truly proactive instead of reactive behavior. As an abstract moving power, each agent is equipped with a set of motives, representing its propensity to achieve certain goals. These motives are used to generate concrete desires together with parameters that represent a degree of intensity of the respective desire according to the agents current situation. Both the generation of new desires and the assignment of parameters are specified by extended logic programs. Using the intensity parameter as a level of motivation supports the deliberation component when deciding which desire should be selected as a goal. It should be mentioned that a single abstract motive can result in different concrete desires. The motive of having cows in the corral in order to win, for example, can result in desires for driving different herds of cows with varying intensities. Likewise, the motive of helping other team members in order to be social can result in different desires, according to the information an agent has about other agents needing help.

Beside selecting which desire the agent should raise to a goal, the deliberation component has to determine the agents intentions in order to fulfill these goals. For that purpose, static and dynamically generated plans are used to cover different cases. In order to adhere to conventions and the phase model described in Section 3.3, static AgentSpeak plans are used to specify roughly the agents' behavior. To generate dynamic plans depending on the agents current situation, the deliberation component makes use of a dynamic planner which is based on $\mathcal{K}$ (see Section 2.3).

At the end of each BDI cycle, if the current intention contains at least one atomic intention, the action selection sends this atomic intention to the environment. In this case, the intention is translated into the corresponding action that can be sent to the simulation server.

### 4.2 Belief Revision

One particular important component in our extended BDI architecture is the belief revision function. A belief revision function has to integrate new information received from the environment or other agents into an agent's beliefs, while keeping an agent's beliefs consistent. In our BDI model, shown in Figure 3, the belief revision component consists of a function labelled BRF and an epistemic state labelled *Beliefs*, where an epistemic state is a structure that holds all necessary data to derive a belief set (a consistent set of facts represented by literals in this case).

We will first give the abstract specifications of belief revision functions and epistemic states and go into structural details afterwards. This specification is loosely inspired by [10]. The atomic piece of information we consider is called an *information object*. An *information object* $I$ is a tuple $(P, Meta)$, where $P$ is an extended logic program and $Meta$ contains additional informations like a source identifier and a time stamp. Let $\mathcal{IO}$ denote the set of all information objects.

**Definition 4 (Belief Base).** *A* belief base $BB$ *is a sequence of information objects* $I$*:* $BB = \{I_0, \ldots, I_n\}, n \in N$*. The set of all belief bases is denoted by* $\mathcal{BB}$*.*

**Definition 5 (Belief Change Operator).** *A* belief change operator *is a function* $\mathcal{BB} \times \mathcal{IO} \rightarrow \mathcal{BB}$*.*

For example, our common belief change operation is to add the program of an information object to the current belief base and ignoring the extra-logical information. Hence. it would be a function like

$$f_{expand}(B, (P, Meta)) = B \cup \{P\}$$

for a belief base $B$ and an information object $I = (P, Meta)$.

**Definition 6 (Argonauts Epistemic State).** *An epistemic state is a tuple* $(BB, P^*, BS)$ *where* $BB$ *is a belief base,* $P^*$ *is a consistent extended logic program, and* $BS$ *is a belief set.*

Let $Bel$ be the set of all epistemic states and $Per$ the set of all perceptions.

**Definition 7 (Belief Revision Function).** *A belief revision function is a function* $Bel \times Per \rightarrow Bel$*.*

The ARGONAUTS belief revision function transforms an agent's epistemic state into another epistemic state. In our approach, beliefs are represented basically by extended logic programs, hence perceptions are logic programs, and belief operations are carried out on logic programs. In order to resolve conflicts between programs, our belief revision operations are based on update sequences as proposed by Eiter et. al [7]. As shown in Figure 4, different functions and data structures are involved in updating an agent's epistemic state and revising its beliefs. We distinguish between belief change operations which are responsible

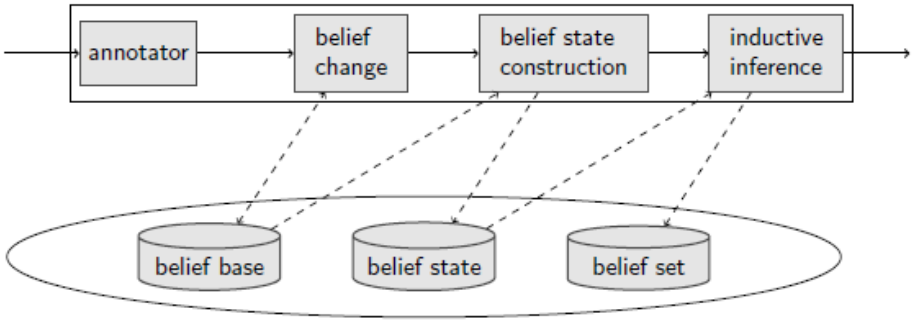


**Fig. 4.** A detailed view of the belief revision component, showing the functions and data structures involved in revising an agent's beliefs

for storing new information, and inference operations which generate a (new) consistent belief set.

Whenever new pieces of information $P \in Per$ are received, the annotator transforms these into information objects. These information objects are integrated into an agent's belief base by a belief change operator.

Besides belief change operations which modify an agent's belief base our agents are capable of performing advanced epistemic operations to generate their beliefs. This is done in two steps: first, a belief state $P^*$ is constructed from information objects stored inside the belief base. Since extended logic programs are the basic ingredients for information objects, belief states are modeled by (consistent) extended logic programs as well. The construction of a consistent extended logic program involves a transformation of each extended logic program from any information object. Basically, each rule from any program used to derive a literal $A$ is rewritten to not fire in the presence of $\neg A$ derived from some other rule. For more details and examples, see [7]. The decision which rule $r_i$ to suppress in presence of a conflicting rule $r_j$ from two programs from two different information objects is based on a priority index. This priority index is based on a total order assigned to all information objects in a belief base, and newer information is prioritized over older information. Second, an answer set solver is used to generate answer sets of $P^*$, and one answer set is propagated to the agent's *belief set*. This is done by operators called inductive inference operators. This process is computational expensive, as the computation of an answer set of an extended logic program is in general NP-complete. This can lead to problems, as an agent might not be able to keep up with the change of world because his belief computations consume too much time.

**Definition 8 (Belief State Construction).** *A* belief state construction *is a function* $\mathcal{BB} \rightarrow \mathcal{P}$.

**Definition 9 (Inductive Inference).** *An* inductive inference operator *is a function* $\mathcal{P} \rightarrow 2^{\mathcal{L}}$.

Note that our belief change operators can be simple (as the common change operator which only expands a belief base in a set theoretical way) since the complex revision procedure is principally performed by state construction and inductive inference.

### 4.3 Motivated Desire Generation

The *desire generation*, named *DesGen* in Figure 3 is based on the approach described in [11]. It uses an agent's specific but static set of motives to generate new *desires* and adjust the degrees of intensity of both the new and the existing ones.

**Definition 10 (Desire Generation).** *Let $Bel$ be the set of all epistemic states, $Des$ be the set of all desire states, and $Mot$ be the set of all motive sets. The* desire generation *is a function of the form $Bel \times Mot \times Des \rightarrow Des$.*

Basically, the desire generation is processed in two steps. First, based on the current beliefs, all motives in the agent's motive set are evaluated, to generate new desires which arise as instantiated literals according to the agent's belief state. These new desires are added to the agent's desire state if necessary. Afterwards, the degrees of intensity of all desires the agent currently holds in its state are adjusted due to the current situation of the agent.

**Definition 11 (Motive).** *Let $Bel$ be the set of all epistemic states and $Des$ be the set of all desire states. A* motive *is a tuple of the form $< n, f_{dgen} >$, with $n$ being the representation of the* motive *by a unique name and $f_{dgen}$ being a function of the form $Bel \rightarrow Des$.*

Representing its abstract moving power, the *motive set* contains all motives of an agent. To generate concrete desires given this abstract power, the function $f_{dgen}$ is used. The desires generated this way are defined as follows:

**Definition 12 (Desire).** *Let $Bel$ be the set of all epistemic states. A* Desire *is a tuple of the form $< id, i, u, m >$, with $id$ being the unique name of the desire, $i \in \mathbb{N}$ indicating the degree of intensity, and functions $u$, $m : Bel \rightarrow \mathbb{N}$ computing* updating *and* mitigation *of the desire. The function $u$ is called the* intensity update function *and $m$ is called* mitigation function.

The two functions $u$ and $m$ are used to adjust a desire's degree of intensity $i$ as its level of motivation. The motivation induces a complete preorder over the desires in the agents *desire state*. The function $u$ is used to increase or decrease the degree of intensity based on the agents current situation. The second function, $m$, mitigates substantially the intensity in case of the fulfillment of the corresponding goal. In order to prevent the execution of nearly unmotivated desires, those with an intensity below a given threshold can be removed from the desire state.

*Example 6.* We now demonstrate how the desire generation works using the example of driving cows into the corral. The abstract motive stands for the force animating the agent to drive cows into its own corral in order to win the match. This motive is defined as

$$< \texttt{cowsInCorral}, f_{dgen}(Bel) >,$$

where $f_{dgen}(Bel) =< drivenCow(X), 0, u, m >$ for some variable `X` with $\texttt{cow(X)} \wedge \neg\texttt{inCorral(X)} \in Bel$. For each such cow `X` for which the agent knows that is not in the agent's corral, a desire is generated. The desire's update function $u$ is used to higher the intensity in case the cow comes close enough, and to slowly lower it if it gets to far away.

$$u(Bel) = \begin{cases} 5 & \text{if } \texttt{close(X)} \in Bel \text{ for the given } \texttt{X} \\ -3 & \text{if } \texttt{faraway(X)} \in Bel \text{ for the given } \texttt{X} \end{cases}$$

In case the cow was driven into the agent's corral the intensity of the corresponding desire has to be mitigated by the function $m$ which is defined as follows:

$$m(Bel) = -30 \quad \text{if } \texttt{inCorral(X)} \in Bel \text{ for the given } \texttt{X}$$

If the agent sees a cow, the motive is used to generate a desire to drive this cow into the corral. Once generated, the functions for updating and mitigating the desires intensity are evaluated each time the desire generation is executed during the BDI cycle. The intensity will rise as the agent comes close enough, and sink if it gets to far away. In case the cow is driven into the corral, the intensity will be mitigated since the desire has been satisfied.

### 4.4 Interaction and Communication

As in the analysis phase, we also made use of the GAIA methodology in the designing phase of the multi-agent system. In order to obtain maximal flexibility our first approach was to implement all roles in only one single agent type. Unfortunately, this led to limited benefit from using GAIA, especially the relationship model became useless. However, it helped to build a list of services each role should provide, which came in use while building the static plans mentioned above.

Since the overall task is to drive herds of cows into the own corral which due to the speed of the cows should be done by more than only a single agent, all tasks are done by a group of agents. This should avoid unnecessary waste of time while waiting for other agents when a herd was found by a smaller scouting group or a single agent. Nevertheless, each agent acts autonomously, driven by its motives. To perform tasks, the agents form dynamic groups and negotiate about the leader role. During task execution, all agents can adopt all other roles, according to the current needs communicated by the group leader.

All communication is done by sending facts to one or more agents. There is no explicitly used speech act theory, yet this is implicitly determined by the semantics of the communicated information. For instance, a communicated meeting point can be understood as the request to *achieve* the adjustment of an agent's current position.

## 5 Programming Language and Execution Platform

In this section we describe the methods and tools we employed for the implementation of our agents and our multi-agent system. We first describe how individual agents are implemented within our MAS and then describe the overall system.

We used JAVA as the common programming language for our agent framework, with Eclipse[2] being the development environment. In addition, JASON [2] was used as the foundation to our agent framework. As we made use of extended logic programs for knowledge representation and reasoning, we incorporated an answer set solver into our framework, in this case DLV [4].

Each agent consists of two parts: at least one AGENTSPEAK plan, and a set of extended logic programs. The plans are mainly used to allow an agent to be executed within JASON and to make use of our modifications and extensions to JASON written in JAVA. The extended logic programs are used to describe and implement belief operations an agent is capable of. In JASON, agents are realized in form of AGENTSPEAK files. Such files consist of a number of beliefs and goals, that represent plans that can be executed in some iterative way. This is called an intention in JASON. We use those plans to specify which coarse actions agents should commit to based upon their motivations. So for every desire which induces a certain role, a plan with an achievement goal matching that desire is processed. In terms of mutual exclusive goals the lesser, or no longer, desired goal is dropped.

Using the AGENTSPEAK constructs provided by JASON only allows a reactive behavior, because each plan relies on some external caused belief change events. However, instead of simply invoking plans directly depending on belief changes (through perception or communication), we generate intentions based on motivations. Therefore, the motives of each agent are used to generate desires with a certain intensity based on its beliefs. This intensity allows the deliberation component to decide to raise a desire to a goal. Hereby, we achieve a more proactive behavior. In addition to plans determining the course of action for motivations, we use fixed plans according to the protocols specified before. In particular, at the beginning of every simulation turn, each agent broadcasts its perceptions to every other agent in the agent society. This behavior is independent of any motivation since we see it as a basic ability every cowbot agent possesses. Predefined AGENTSPEAK plans are only used to create a coarse skeleton for our agent behaviors. We use JASON mainly for communication and environment interaction purposes among agents. Most static plans are in general short and intended to call an internal planner like the $\mathcal{K}$ front end of DLV to dynamically generate plans.

Another important part of each agent is the set of extended logic programs. Each extended logic program represents its beliefs about the world in form of facts and rules. For every percept, an agent stores that percept in the extended logic program belief base (not to be confused with the JASON belief base). Then, the belief revision function is applied. The belief revision function uses some

[2] `http://www.eclipse.org`

**Listing 4** Pathfinding Algorithm

```
1: while Q ≠ ∅
2:    take first entry (c, d_c) from queue Q
3:    check four adjacent cells c_nb of cell c with distance value d_c
4:    for every c_nb:
5:       if c_nb has distance value of 0 and is not an obstacle
6:          put (c_nb, d_c + 1) into Q
```

kind of transformation of all extended logic programs within the belief base to generate one consistent logic program. Calling an answer set solver like DLV allows an agent to gain a new set of beliefs, which is mapped to JASON's internal belief base.

## 6 Agent Team Strategy

Since we consider only a single type of agent in the cows and herders scenario every single agent can adopt every needed role. By doing so it can be part of a dynamically formed group that is either looking for or driving a herd of cows.

For both the agents navigation and calculating the herds path to the corral, we implemented a flood fill pathfinding algorithm, which starts filling at the destination of the path. The pathfinding algorithm consists of two steps: setting up a data structure for the search space, and then performing the search itself. In the first step, we setup a distance map, where each entry corresponds to a grid world cell. Each entry contains a number. The number value represents the distance towards a target location, or a special value indicating that there is no distance available. The distance is measured in four-way axis-parallel steps an entity has to walk to reach a certain target location. That target location is the reference point for setting up distance values for other cells, and is initialized with 1. All other entries are set to zero, which indicates that no path to a target location exists. All obstacles are initialized with a very high distance value, which can never be achieved in cycle-free paths over the map. Then, the target location coordinate and distance value is put into a FIFO queue. The algorithm sketched in Listing 4 to compute a distance value for each cell.

In the second step, the path finding is performed, based on information stored in the distance map. If the starting position has a value of 0, no path exists and search is over. Otherwise, a path exists. The path calculation is rather simple. For a given starting location, look at all cells directly reachable by allowed movements (eight-way directions in common, or four-way if an agent is adjacent to a switch). Choose the cell with the smallest distance value smaller than the current location, and continue until a cell with a distance value of one is found. Since the algorithm does not destroy information stored in the distance map, it can also be used for a fast look-up of distances between cows and the corral center, and cluster nearby cows to herds.

Thus, it can easily be determined which herd is closest to the corral and which agent is closest to a herd or a switch. To avoid unnecessary movements if the agents walk in close formation, agents are not seen as obstacles. Fences are not seen as obstacles neither as having the intention to cross the fence implies that another group member will open it. This is ensured by the group leader during the task decomposition. Whenever it turns out that the group has to pass a fence, the group leader will, according to the information other team members communicated, send the agent closest to the switch to open it. In addition to that, the group leader will also try to keep the group together by sending another agent to open the fence on the other side, since it is not possible for a single agent to open a fence and pass it at the same time. To drive a herd of cows, the agents simply align in a row behind the herd in a distance causing the cows to walk in the desired direction. Therefore, a simple formation algorithm is used which takes care of obstacles. We do not provide for an explicit action to obstruct opponent agents or to steal cows from the opponent team. Nevertheless, we decided to modify the motives of some agents, causing them to prefer cows in the enemy corrals. This led to a group of agents trying to steal the opponents cows if in range.

As mentioned above, a group is coordinated by a group leader who is responsible for decomposing the current task into subgoals and assigning them to the other group members while considering which agent can fulfill each goal at the lowest effort. Since the situations are subject to frequent change, only the very next step the group will take is planned. The subgoals assigned to the single agents require individual planning. For instance, an agent given the goal to open a fence has to plan if and where to look for a switch and which way should be taken to the destination. There is no coordination among the individual groups which led to problems especially when two herding groups cross their ways.

Figure 5 shows a snapshot of four agents which currently pursue the goal to drive a herd of cows into their own corral. They passed the first of two fences in their way. Due to the fact that other fences are lying between the agents and the cows a second agent adopted the door-opener role to keep the fence open from the other side. Hence, the first door-opener can close up to the rest of the group. The agent closest to the switch of the second fence is sent to open it. The other agents line up in a row behind the herd of cows to drive them towards the desired direction as mentioned earlier. In this special case, the last fence between the agents and the cow is the same one that crosses the desired way the cows should take. Therefore, the fence will be kept open from only one side without trying to keep the group together by sending a second door opener from the other side.

Initially we planned to exchange only relevant information on team level and even less information on global level. It turned out that far to often important information was missing which resulted in poor plans and behavior of the agents. This is aggravated by the fact that every agent uses its own knowledge to calculate paths, so we decided to exchange all new information an agent gets by its own perceptions.

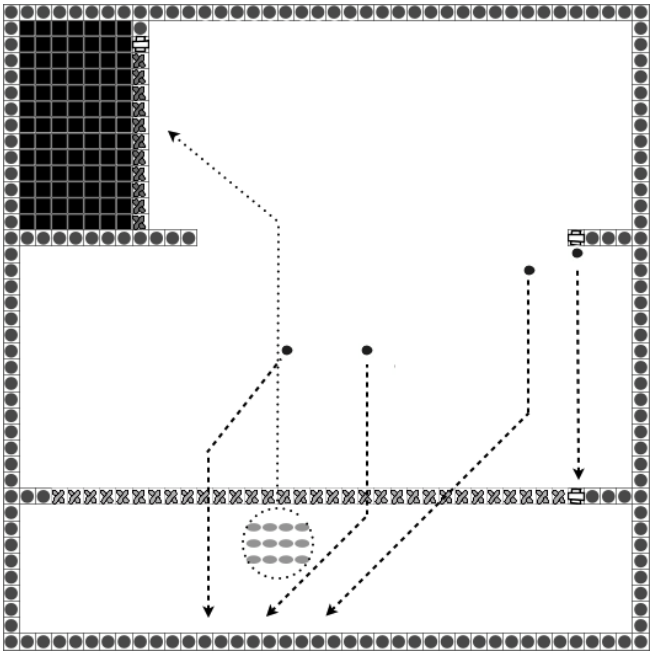

**Fig. 5.** Four agents working in a group to drive a herd of cows.

For future work, besides the coordination within the group, a global coordination should be developed to avoid conflicts between the actions of different groups and to take advantage of side effects caused by several groups with similar goals.

## 7 Technical Details

During development, we used ordinary personal computers for coding and testing the framework. Since our general and open approach required large times for computing, for evaluation purposes, we had to either reduce the number of agents or expand the time intervals during game steps. During the contest we used a dedicated eight-core linux machine that was still not fast enough to ensure that all agents would send their actions during the given interval. Although this had not affected the correct execution of every agents BDI cycle, it would have led to very slowly acting agents especially during group forming, being not competitive in any case. On these grounds, we made a few decisions to increase our systems speed. First, we replaced some generic logical parts of our framework by scenario-specific JAVA equivalents, like a specialized way to handle the agents knowledge of the world. Secondly, we striped off some complex parts in the agents team strategy, like the dynamic group forming, which was replaced by static groups

with designated leaders. This led to an decreased amount of logical rules, and so decreased the execution time of the answer set solver mentioned in Section 4.2.

Apart from these speed issues which will likely be subject to our future work we had no technical problems with the framework. It ran stable during the whole contest, not least because of the possibility to reset each agent's connection to the simulation without restarting the agent itself.

## 8 Discussion and Conclusion

During the first half of our project we designed and implemented a framework for executing extended BDI agents. Although the requirements taken into account during development were partially generated by the cows and herders scenario, it was not tailored for the contest, but rather provides customizability in most components. Using the GAIA methodology we designed a multi-agent system for participating in the contest that builds on top of this framework. Due to the abstract nature of the GAIA methodology we experienced no restrictions for the development of the system.

The contest was not only a great way to fix many bugs and benchmark the framework it was also a great opportunity to design a multi-agent system that has to be competitive. During the matches against other teams with different strategies we gained deep insights into our own framework. Intense discussions on how to improve both the framework and the multi-agent system were set off by the matches. It was a motivating way to learn about multi-agent systems and knowledge engineering. Besides some technical bugs and inconsistencies in the agents strategy the major issue we faced was the insufficient speed of our implementation which to increase will be part of our future work. Since this was the first time we participated in the contest we were happy with the complexity of the scenario. However, we think it could be improved by requiring more long term planning. Furthermore, it should not be possible to harm the opponent so much by simply patrolling in its corral.

## References


1. Bienek, F., Böhmer, E., Broszeit, S., Hölzgen, D., Jablkowski, B., Kruse, M., Löwen, A., Vengels, T.: Endbericht PG 545 – Intelligent Cowbots. Tech. rep., Technische Universität Dortmund, Germany (2010)
2. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology). John Wiley & Sons (2007)
3. Bratman, M.E.: Intention, Plans, and Practical Reason. CSLI Publications (1987)
4. Eiter, T., Faber, W., Koch, C., Leone, N., Pfeifer, G.: DLV - a system for declarative problem solving. In: Baral, C., Truszczynski, M. (eds.) Proc. of the 8th Int. Workshop on Non-Monotonic Reasoning (2000)
5. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: Planning under incomplete knowledge. In: Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L., Sagiv, Y., Stuckey, P. (eds.) Computational Logic -

CL 2000, Lecture Notes in Computer Science, vol. 1861, pp. 807–821. Springer Berlin / Heidelberg (2000)
6. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning: Semantics and complexity. ACM Transactions on Computational Logic 5, 206–263 (April 2004)
7. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: On properties of update sequences based on causal rejection. Theory and Practice of Logic Programming 2, 711–767 (November 2002)
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
9. Gelfond, M., Leone, N.: Logic programming and knowledge representation - the a-prolog perspective. Artificial Intelligence 138(1-2), 3–38 (June 2002)
10. Krümpelmann, P., Thimm, M., Ritterskamp, M., Kern-Isberner, G.: Belief Operations for Motivated BDI Agents. In: Padgham, L., Parkes, D.C., Müller, J.P., Parsons, S. (eds.) Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008). pp. 421–428. Estoril, Portugal (May 2008)
11. Meneguzzi, F.R., Luck, M.: Motivations as an abstraction of meta-level reasoning. In: Burkhard, H.D., Lindemann, G., Verbrugge, R., Varga, L.Z. (eds.) Multi-Agent Systems and Applications V: Proceedings of the Fifth International Central and Eastern European Conference on Multi-Agent Systems. pp. 204–214. No. 4696 in Lecture Notes in Computer Science, Springer (2007)
12. Pokahr, A., Braubach, L.: From a Research to an Industrial-Strength Agent Platform: Jadex V2. In: H. R. Hansen, D. Karagiannis, H.G.F. (ed.) Business Services: Konzepte, Technologien, Anwendungen - 9. Internationale Tagung Wirtschaftsinformatik (WI 2009). pp. 769–778. Österreichische Computer Gesellschaft (2009)
13. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First Intl. Conference on Multiagent Systems. San Francisco (1995)
14. Rao, A.S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: de Velde, W.V., Perram, J.W. (eds.) MAAMAW. Lecture Notes in Computer Science, vol. 1038, pp. 42–55. Springer (1996)
15. de Silva, L., Sardina, S., Padgham, L.: First Principles Planning in BDI Systems. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems. pp. 1105–1112 (2009)
16. Weiss, G. (ed.): Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press, Cambridge, MA, USA (1999)
17. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems pp. 285–312 (2000)